

# Intelligent Computer Games



David Levy

Correspondence is welcome. Letters with interesting questions and ideas will be used in the column along with a response. No personal replies can be made. Send to: David Levy, 104 Hamilton Terrace, London NW8 9UP, England

## Two-Person Games

Two-person games, such as chess, backgammon and checkers, are usually more interesting and challenging than one-person games, and it is to these that we shall be devoting most of our studies. The introduction of a second player creates manifold difficulties that do not exist in a one-person game, but fortunately for today's programmers these difficulties have been extensively analyzed in the computing literature and the problems are now rather well understood.

## The Two-Person Game Tree

Game trees become more complex structures when an opponent appears on the scene. Let us consider a relatively simple game, noughts and crosses (tic-tac-toe to our American cousins), and examine how its tree will

look after a move or two of look-ahead. We shall assume that "cross" moves first.

- From the initial position there are three essentially different moves:
- 1) e (the centre)
  - 2) a,c,g, and i (the corners)
  - 3) b,d,f, and h (middle of the edges)

On the first move, any of group 2 is equivalent to any other, since all four moves are merely reflections or rotations of each other. Similarly, within group 3 all moves are equivalent. This technique of utilizing symmetry to reduce the magnitude of the problem is worthwhile when programming a game that lends itself to a symmetrical analysis. By reducing the number of moves that need to be examined at any point in the tree you will be cutting execution time dramatically, because the combinatorial effects of tree growth are enormous. The savings in time that can be achieved through using symmetry can be extremely valuable when improving the performance of the program by making its evaluation function more sophisticated (and slower).

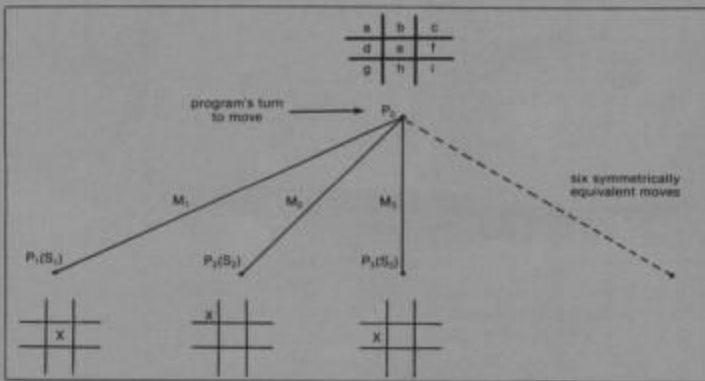
If we so decide, our program can terminate its search of the tree after looking at each of its possible moves from the root. This is called a 1-ply

search because the program only looks one "ply" deep. (The term "ply" is used to denote a single move by one player.) In order to decide which move to make, out of  $m_1$ ,  $m_2$ , and  $m_3$ , the program will then apply its evaluation function to the three positions at the lower end of the tree (these are called the terminal positions). Whichever position had the best score will then be assumed to be the most desirable position for the program, and the program will make the move leading to that position.

How should we set about designing our evaluation function? This is one of the fundamental problems in game playing programming because a good evaluation function will help the program to make good judgements, and hence to play well, even though the depth of look-ahead may be shallow. A poor function, on the other hand, might well result in poor play even with a deep and time-consuming search of the game tree. It is therefore worthwhile to put some careful thought into the design of the evaluation function, and the following example should illustrate the type of thinking that is necessary.

The object of the game is to create a row of three of your own symbols. We shall call this a "3-row." The next most important thing is to prevent your opponent from making a 3-row, which means that he should not have a 2-row after you move (a 2-row has two symbols of one player and one empty space). Next most important is the creation of your own 2-rows; then it is important not to leave your opponent with 1-rows (one of his symbols and two empty spaces); and finally you should try to create your own 1-rows. All of these features could well be incorporated into a noughts and crosses evaluation function.

If we denote the number of cross' 3-rows by  $c_3$ , the number of nought's 2-rows by  $n_2$ , the number of cross' 2-rows by  $c_2$ , the number of nought's 1-rows by  $n_1$ , and the number of cross' 1-rows by  $c_1$ , then one measure of the merit of a



## Games, cont'd . . .

position from cross' point of view would be

$$C_3 - n_2 + C_2 - n_1 + C_1$$

but this measure has one obvious drawback. It does not allow for the fact that the term  $C_3$  is more important than  $n_2$ , which is more important than  $C_2$ , and so on. This can be done by multiplying each of the terms in the evaluation function by some numerical weighting, in such a way that the weightings (hopefully) reflect the relative importance of each feature. The evaluation function then becomes

$$(k_3 \times c_3) - (k_2' \times n_2) + (k_2 \times c_2) - (k_1' \times n_1) + (k_1 \times c_1)$$

where  $k_3$ ,  $k_2'$ ,  $k_2$ ,  $k_1'$  and  $k_1$  are the numerical weightings. Since one  $c_3$  is worth more than all the  $n_2$ 's in the world, i.e., a winning row is more important than any number of 2-rows, we can set  $k_3$  to be some arbitrarily high number, say 128. By studying the game for a few minutes it is possible to see that if one side has a 3-row, the other side may have at most two 2-rows, so to reflect the relative importance of one's own 3-rows and enemy 2-rows it is necessary to ensure that  $k_3 > 2 \times k_2'$ . We can therefore try  $k_2' = 63$ . (If one side has a 3-row and his opponent two 2-rows, the opponent will not have any 1-rows to upset this scoring mechanism.)

If there are no 3-rows, but one side only has a 2-row, his opponent cannot have more than three 1-rows, as in the following situation:

	0	
X		0
X		

$$\text{So } k_2' > 2 \times k_1 \text{ and } k_2 > 2 \times k_1'$$

and we can try  $k_2 = 31$ ,  $k_1' = 15$  and  $k_1 = 7$ . Remember that we can modify these values in the light of experience with the program, the values 128, 63, 31, 15 and 7 are merely our first estimates. Having made these estimates we should then ensure that the score for a noughts and crosses position will never cause an overflow, and we do this by setting up positions which will have the largest and smallest possible scores, and counting the number of 3-rows etc. in each. This is a very important part of evaluation function design, and I remember a chess programmer who could not understand why his program crashed whenever it was winning or losing by a great margin — he had forgotten to allow for the possibility of one side being two queens ahead and when that happened his evaluation calculations created an overflow.

If we now return to Figure 1 we can

see that each of the three possible first moves results in the creation of a different number of 1-rows. Applying the evaluation function

$$128 \times c_3 - 63 \times n_2 + 31 \times c_2 - 15 \times n_1 + 7 \times c_1$$

to the three positions  $P_1$ ,  $P_2$  and  $P_3$  we find that in each case  $c_3 = n_2 = c_2 = n_1 = 0$ , and therefore:

$$S_1 = 128 \times 0 - 63 \times 0 + 31 \times 0 - 15 \times 0 + 7 \times 4 = 28$$

$$S_2 = 128 \times 0 - 63 \times 0 + 31 \times 0 - 15 \times 0 + 7 \times 3 = 21$$

$$S_3 = 128 \times 0 - 63 \times 0 + 31 \times 0 - 15 \times 0 + 7 \times 2 = 14$$

and  $S_1$  is the most desirable of these scores so the program would make the move  $m_1$  to reach position  $P_1$  (i.e. it would play in the center).

### The 2-Ply Search

The 1-ply search is the simplest form of tree search in a two-person game, but it does not take into account the fact that once the program has made its move there is an opponent waiting to reply. It may be the case that a move which, superficially, looks strong, is seen to be an error when we look a little bit further into what may happen. The 2-ply search will "see" more than the 1-ply search and so moves made on the basis of a 2-ply search will be more accurate (provided the evaluation function is not a disaster area). How can we take into account this extra dimension of the opponent's move?

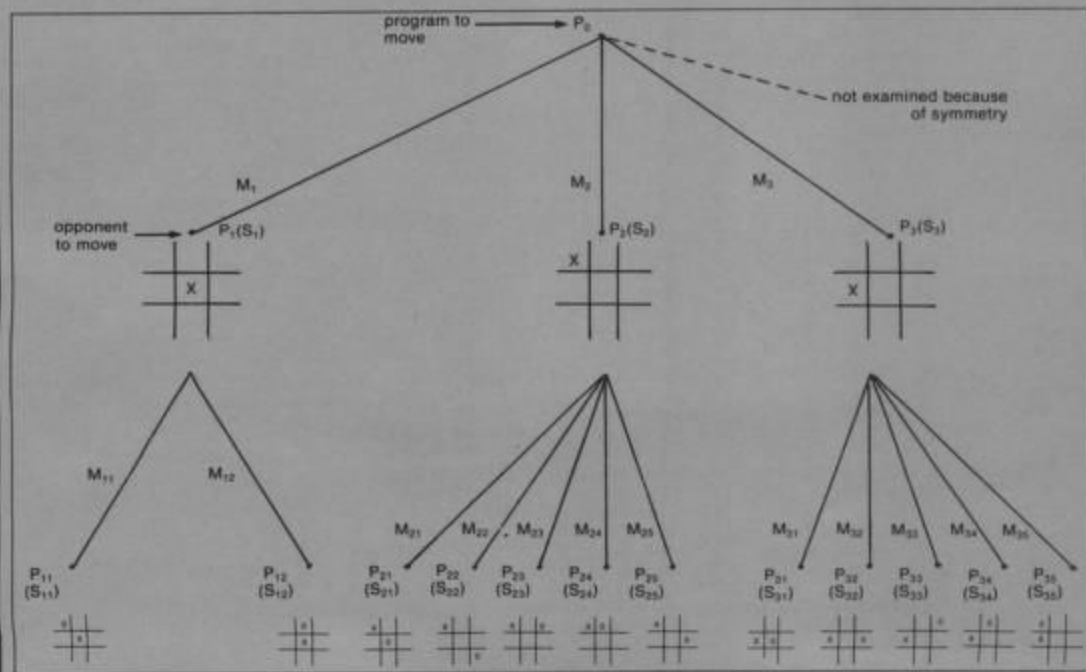


FIGURE 2.

## Games, cont'd . . .

Let us look at the same tree, grown one ply deeper, i.e., to a total depth of two ply — one move by the program and one move by its opponent.

If "cross" plays in the center, "nought" has two essentially different replies, in a corner or on the middle of an edge (represented by positions  $P_{21}$  and  $P_{12}$ , respectively). If "cross" makes his first move in a corner ( $P_2$ ), "nought" will have five different reply moves ( $m_{21}$ ,  $m_{22}$ ,  $m_{23}$ ,  $m_{24}$  and  $m_{25}$ ) leading to positions  $P_{31}$ ,  $P_{32}$ ,  $P_{33}$ ,  $P_{34}$  and  $P_{35}$ . After "cross" plays move  $m_3$ , "nought" again has five replies. It is easy to see how the tree grows. In last month's example, the 8-puzzle, the *branching factor* (number of branches from each position on the tree) was never more than three. Here it is more, even allowing for symmetry.

Let us consider how the program might analyze the situation. It uses its evaluation function to assign scores to the terminal nodes  $P_{11}$  and  $P_{12}$ . In each case  $c_2 = n_2 = c_1 = 0$ . In position  $P_{11}$ ,  $c_1 = 3$  and  $n_1 = 2$ . In position  $P_{12}$ ,  $c_1 = 3$  and  $n_1 = 1$ .

We now have:

$$S_{11} = (-15 \times 2) + (7 \times 3) = -9$$
$$S_{12} = (-15 \times 1) + (7 \times 3) = 6$$

This information indicates that if the program is sitting in position  $P_1$ , with its opponent to move, its opponent may choose between moves  $m_{11}$  (leading to position  $P_{11}$  of value -9) and  $m_{12}$  (leading to position  $P_{12}$  of value 6). The program's opponent wants to minimize the score and so it would choose move  $m_{11}$ , for a score of -9, and so the real value of position  $P_1$ , represented by  $S_1$ , is this *backed-up* score of -9.

If we apply the evaluation function to positions  $P_{21} \dots P_{25}$  we will get:

$$S_{21} = (-15 \times 3) + (7 \times 2) = -31$$
$$S_{22} = (-15 \times 2) + (7 \times 2) = -16$$
$$S_{23} = (-15 \times 2) + (7 \times 2) = -16$$
$$S_{24} = (-15 \times 1) + (7 \times 2) = -1$$
$$S_{25} = (-15 \times 2) + (7 \times 3) = -9$$

Wishing to minimize the score when making its move from  $P_2$ , the program's opponent would choose move  $m_{21}$ , leading to position  $P_{21}$  and a score of -31.

Similarly, when applying the evaluation function to positions  $P_{31} \dots P_{35}$ , we get:

$$S_{31} = -38$$
$$S_{32} = -8$$
$$S_{33} = -31$$
$$S_{34} = -16$$
$$S_{35} = -23$$

so the program's opponent, when making its move from  $P_3$ , would choose move  $m_{31}$  for a score of -38.

We now have the following situation. If the program makes move  $m_1$ , its opponent, with best play, can achieve a score of -9. If the program plays  $m_2$ , then its opponent can achieve a score of -31. If the program plays  $m_3$ , then its opponent can score -38.

Just as the program's opponent wishes to minimize the score, so the program wishes to maximize the score. The program must now choose between  $m_1$  (for -9),  $m_2$  (for -31) and  $m_3$  (for -38). Since the maximum of these three values is -9, the program will play move  $m_1$ , and the backed-up score at the root of the tree will be -9. This represents the score that will be achieved with best play from both sides.

This procedure of choosing the maximum of the minimums of the maximums of the minimums . . . etc. is known, not surprisingly, as the *minimax* method of tree searching. It is an algorithm that finds the move which will be best, assuming correct play for both sides, providing that the evaluation function is reasonably accurate.

### Memory Requirements for a Minimax Search

One of the great advantages of the minimax type of search is that it is not necessary to retain the whole tree in memory. In fact it is necessary to keep only one position at each level of look ahead, together with a certain amount of information about the moves from each of these positions. Let us see how this works for our 2-ply tree.

From the initial position  $P_0$ , the program generates the first move for cross, to position  $P_1$ . Before proceeding to the other moves that cross can make, the program generates the first reply move by nought,  $m_{11}$ , reaches position  $P_{11}$  and assigns it the score  $S_{11}$  (-9). This is the first terminal node to be evaluated, so the score of -9 represents the best score found so far and this is the score that is assigned to  $S_1$ . Since  $P_1$  is the first move at 1-ply to be examined, this score of -9 also represents the best score found so far at the 1-level, and this is the score assigned to  $S_0$ .

The program now looks at  $P_{12}$ , which we sometimes refer to as the brother of  $P_{11}$  (and  $P_1$  is father to both of them). The program determines the score  $S_{12}$ , compares this value (6) with the best score found so far at this level (-9) and finds the -9 preferable, so the scores  $S_1$  and  $S_0$  need not be adjusted at this stage. The program next looks for another brother to  $P_{11}$ , but finding none, it goes back up the tree and looks for a brother to  $P_1$ , which leads it to position  $P_2$  and then to  $P_{21}$ . On the way down this part of the tree

the program assigns to  $P_2$  a score of -9, since this is the best that can be achieved so far. When looking at  $P_{21}$ , the program finds a score of -31, which is better for the program's opponent than -9 and so  $S_2$  is now set to -31.

Note that as this process continues, the brother nodes that have been examined in the past no longer serve any useful purpose and so they can be discarded. At the present point in our search we no longer need the brother of  $P_2$  that has already been examined ( $P_1$ ), so  $P_1$  and its successor nodes are not kept in the tree at this time. The tree, at this moment, comprises only  $P_0$ ,  $P_2$  and  $P_{21}$ .

Having evaluated  $P_{21}$ , we throw it away and look at  $P_{22}$ , which has a score of -16. The program's opponent would not prefer this to the -31 already discovered, and so no change is made to  $S_2$ . The program discards  $P_{22}$  and replaces it with  $P_{23}$  for a score of -16, also of no value to the program's opponent, and this is replaced in turn with  $P_{24}$  and  $P_{25}$  which also produce no change in  $S_2$ . Since  $S_2$  (-31) is less attractive for the program than the best score found so far (-9 at  $S_0$ ), the score at  $P_2$  is not backed-up.  $P_2$  itself is discarded to make way for  $P_3$ , and the same process continues, with the program looking in turn at the scores of  $P_{31} \dots P_{35}$ .

### Task for the Month

The evaluation function for noughts and crosses which we have been using in this example has five features. Try to devise evaluation functions with as few features as possible, for playing noughts and crosses with (a) a 2-ply search; and (b) a 3-ply search, and test your functions by writing a program to play the game using a minimax search. The fact that deeper search will sometimes compensate for a less powerful evaluation function may make it possible for you to reduce the number of features while still writing a program that can play perfectly. If you complete this task, or even if you do not, you might like to think of a way to make the search much faster. This will be the subject of next month's article. □

